AUTOMATIC CREATION OF A CODE GENERATOR FROM A MACHINE DESCRIPTION

Perry L. Miller

May 1971

PROJECT MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Cambridge Massachusetts 02139

ACKNOWLEDGEMENTS

Work reported herein was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defence, under Office Research Contract Number Nonr-4102(01).

I would also like to express my appreciation to Prof. Robert M. Graham who originally suggested this area for research and who helped me in getting started and directed in my efforts, to Prof. Arthur Evans for his continued and cheerful advice and supervision, and to Carla Vogt, Sterling Eanes, and Robert Goldberg for their helpful criticism and comments during the writing of this paper.

AUTOMATIC CREATION OF A CODE GENERATOR FROM A MACHINE DESCRIPTION*

Abstract

This paper studies some of the problems involved in attaining machine independence for a code generator, similar to the language independence and the token independence attained by automatic parsing and automatic lexical systems. In particular, the paper examines the logic involved in two areas of code generation: computation and data reference. It presents models embodying the logic of each area and demonstrates how the models can be filled out by descriptive information about a particular machine. The paper also describes how the models can be incorporated into a descriptive macro code generating system (DMACS) to be used as a tool by a language implementer in creating a machine independent code generator, which can be made machine-directed by a suitable description of a particular machine.

*This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering, Massachusetts Institute of Technology, in partial fulfillment of the requirements for the degree of Electrical Engineer, March 1971.

CONTENTS

CHAPTER I

1.1 INTRODUCTION

The process of translating a high level language into machine instructions is traditionally divided into three distinct problems: lexical analysis, syntactic analysis, and code generation. The flow of data in such a translator is outlined in Figure 1.1.

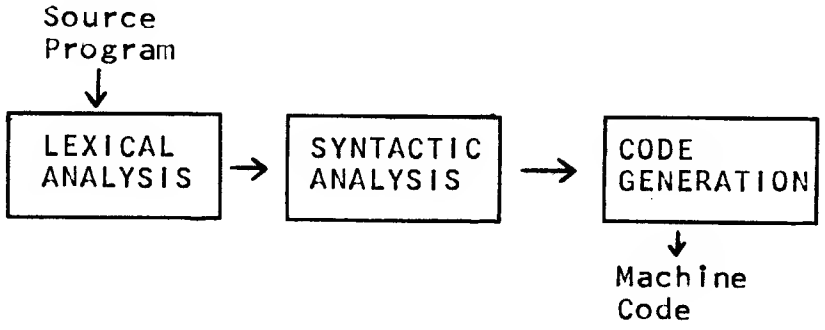


Figure 1.1: Simple Diagram of a Compiler

The lexical analyzer accepts a string of characters and groups these into identifiers and operators, etc., thus creating a string of lexical 'tokens'. The parser analyzes the underlying syntactic structure of this string of tokens, outputting either a sequence of macro operations or a parse tree. The code generator then translates the macros (or the parse tree structure) into machine instructions for

a particular target machine.

Both lexical analysis and syntactic analysis have been intensively studied. Johnson et al. (4) describe a system which allows a lexical analyzer to be automatically created from a series of regular expressions describing possible input lexical tokens. Similarly, numerous parsing schemes (1,2,3) have been developed which allow parsers of varying power to be created automatically from a context-free BNF description of a language. Very little work, however, has been done to similarly formalize and automate code generation. The present research represents an attempt to isolate some of the problems involved in code generation and to show how a code generator can be automatically created from a description of the computer upon which the code is to be run.

The research does not attack all the problems that such an automatic code generating system would have to handle. Rather, it deals with two subproblems corresponding to two common types of macro, namely:

1. computational macros, such as ADD, MULTIPLY, OR, etc.;
2. data reference, such as subscripting and structure

reference.

In this paper, we examine both types of macro in turn and develop a model for the logic of such a macro. We then show how a system can be set up to perform the machine dependent part of such macro logic from machine descriptive information.

The two models developed for the operation of the two types of macro are different. As a result, the paper can be considered to contain two relatively independent topics: the first dealing with computational macros, and the second dealing with data reference macros.

1.2 PREVIOUS WORK

Although little work has been done to formalize code generation, a great deal of work has been done on the related problem of language transferability. One approach to this problem is that of the 'mobile programming system' of Orgass and Waite. (5,6) In their system, the source language is translated into a series of simple macros. Then a user-written set of macro definitions translates the macros into machine code. The problem of generating code for a new machine reduces to the problem of recoding the

macro definitions.

A second approach to language transferability is that of the UNCOL macro language (7,8). UNCOL (UNiversal Computer Oriented Language) was developed in an attempt to create a universal macro language into which all high-level languages could be translated and which itself could be translated into any machine code. If sucessful, the UNCOL system would have solved the problem of language transferability, since only one translator would ever have to be written for a language, and only one code generator for a machine. The Orgass and Waite system differs from the UNCOL approach in that their macro language was specifically tailored to their source language. In practice, the restriction imposed by having only one intermediate language for all source languages and all machines has proven too confining for a practical solution.

The two systems just described are similar in that both attempted to solve the problem of language transferability by letting the user specify information about his machine in procedural form. Most of the information about machine structure is buried implicitly in the coding of the macros. Such a procedural approach has

been used in all major published work on code generation. In contrast, the present work uses information about machine structure given in explicit, descriptive form.

1.3 BRIEF HISTORY OF CODE GENERATION

Early languages had very few data types. Fortran, for example, had only two data types. Similarly, early machines tended to have a small number of special-purpose registers. For such language-machine pairs, the process of generating code tended to be straight forward. A macro generally consisted of a short, independent section of logic which performed a few simple tests and then output code. Thus a very simple procedural language could let the user define these macros (12).

With the introduction of more complicated machines and of languages with more data types, some of which (such as bit-strings) may be more complicated, code generation has become a harder task (9,13). Separate modules have become desirable to handle register manipulation and to handle data-dependent logic for the various data types. Such a modular approach allows a macro to be written fairly compactly, calling these modules as subroutines to locate free registers and to return usable representations of

operands (such as a displacement and registers containing a base and an index).

In a traditional macro system, all of these modules and macros must be written by the user using a procedural language provided for the purpose. Due to the complexity of modern languages and machines, such a macro language can no longer be a very simple one. Similarly, the job of writing a code generator is much more complex.

1.4 DMACS: A DESCRIPTIVE MACRO SYSTEM

This paper describes an automatic code generating system named DMACS. There are two steps in creating a code-generator using DMACS. The first step is to define a set of procedural macros in a machine independent, somewhat skeletal form. The second step is to supply information describing the computer for which code is to be generated. DMACS uses this information to flesh out the macro definitions. The two steps are quite independent, so that once the first step is done for a language, the second step can then be done for a variety of object machines. Similarly, once a machine has been described, implementing a second language requires little change to the machine

description.

The first step can be thought of as defining the semantics of the language using machine independent primitives. The second step can be thought of as defining the structure of the target machine. Examples of the two steps are discussed in Chapters 3 and 4. To facilitate these two steps, DMACS provides two languages:

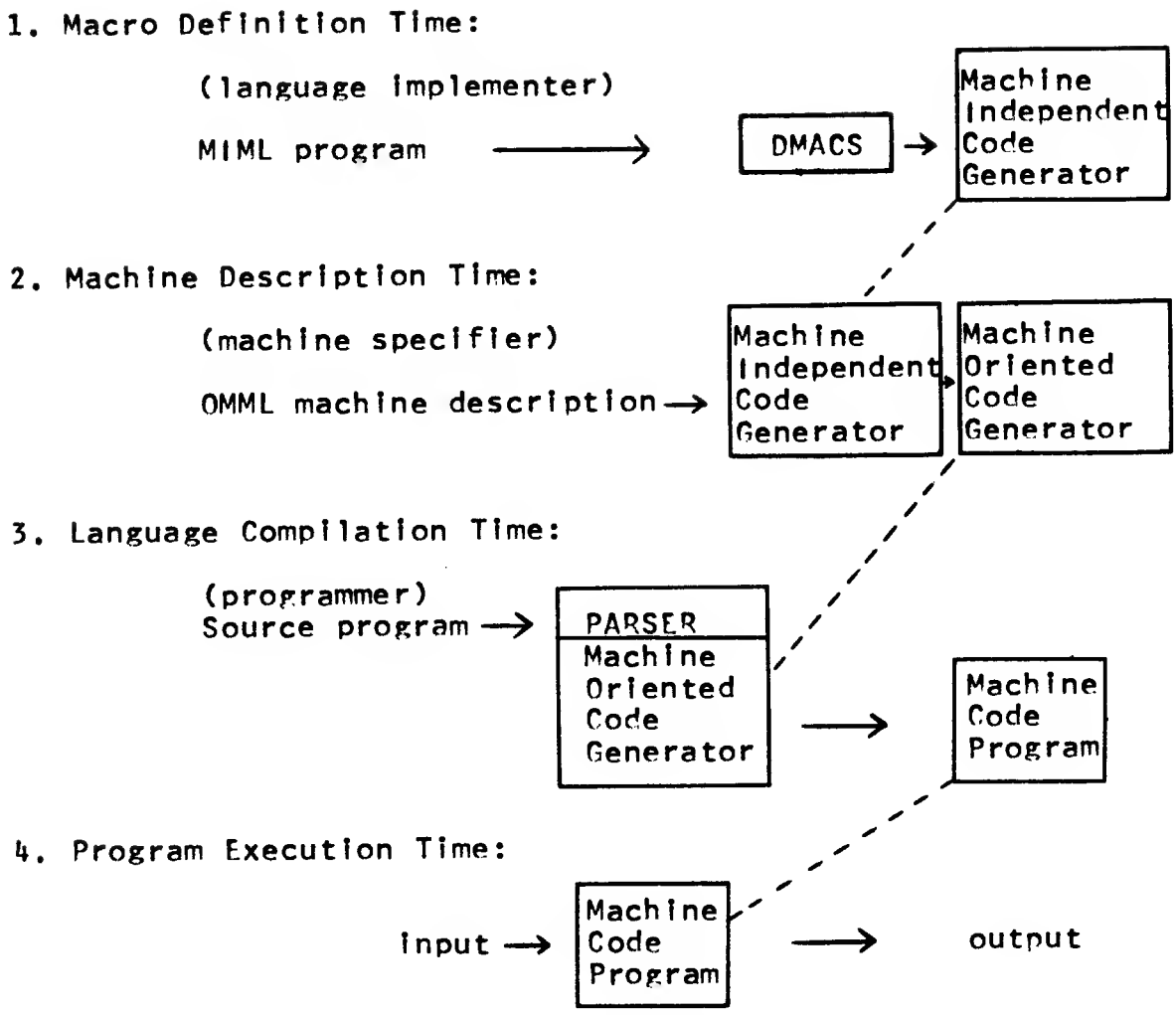
1. MIML- a procedural machine independent macro language, and
2. OMML- a declarative object machine macro language.

Programs written in the two languages are bound together by the DMACS system.

Figure 1.2 outlines how the DMACS system is used. As can be seen, the traditional compile-time vs. run-time distinction has proliferated into four separate 'times' in viewing DMACS as a whole.

1. Macro definition time- when a language implementer presents his machine independent macros to DMACS.

2. Machine description time- when a machine specifier inputs a description of his machine to fill out the machine independent macros.



Using DMACS: 4 Users and 4 'Times'

Figure 1.2

3. Language compilation time- when a programmer inputs his source program to the compiler as a whole.

4. Program execution time- when that compiled program is actually executed.

1.5 OVERVIEW

The present research develops models of two types of macros: computation and data reference macros. At the same time, the paper illustrates how these models can be built into DMACS as tools. These tools can be used by a language implementer to create machine independent macros defining the semantics of his language which can be filled out from a machine description.

Chapter 2 gives the reader an overall introduction to code generation and to the DMACS system. It also discusses some of the restrictions as to possible machine structure which are assumed in the following chapters.

Chapter 3 presents a model of the logic of computational macros. The model pictures a code generator as a state machine whose state is determined by the location of the values used in generating code. In the model, each

computational macro has 'permitted' states for its operands, from which code can be emitted. For the IBM-360, for instance, the permitted states for integer addition would allow both operands in registers or one operand in a register and the other in a word of core memory. To generate code for such a macro, the code generator must make a transition into a permitted state and then emit an appropriate instruction sequence from that state.

Using a procedural macro system, the user specifies how such state transitions are to be made. In a descriptive system such as DMACS, the transitions must be performed automatically from a description of the register and memory structure of a machine, and of the paths (load, store, register-register transfers) between core memory and registers.

Chapter 4 turns to the problem of achieving the same machine independence for data reference macros. To achieve this goal, a data definition facility is built into DMACS. The language implementer writes his data reference logic in terms of the primitives of the facility. A machine specifier then describes his machine memory and how source data items are mapped into that memory. DMACS can then

characterize these source data items in terms of the primitives of the data definition facility. As a result, the macro logic is able to operate upon them.

In summary, the research is a step towards creating models of two aspects of the code generation process, and towards abstracting code generation from any particular machine. In this paper we show how these models can be implemented as tools to be used by a language implementer to create a machine independent code generator which can be filled out from a machine description. Furthermore, it is seen that this approach to code generation, as a natural by-product, leads to a clean separation of the semantics of a source language from the structure of a particular target machine, a separation which is often hard to isolate in a compiler with a code generator oriented towards a particular machine.

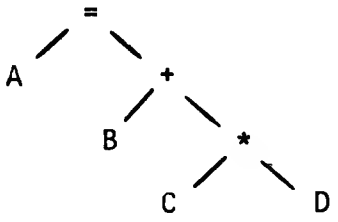
CHAPTER II

A DESCRIPTION OF A CODE GENERATOR

2.1 INTRODUCTION TO CODE GENERATION

Code generation is the last major task in the translation of a high-level language into machine language. A code generator receives its input from the syntactic analyzer (the parser). Although in some compilers the input is in the form of a parse tree, in this paper it is assumed that the input is in the form of a linear sequence of macro operations.

```
A = B + C * D;
```



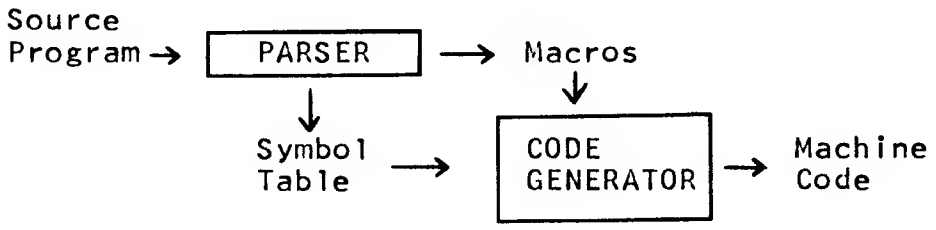
Parse Tree

```
1     MUL   C,D
2     ADD   1,B
3     ASSG  A,2
```

Macros

This assumption is not a restriction, however, since a parse tree can readily be converted into such a sequence of macros. The task of the code generator is to convert the macros into machine instructions.

In a compiler for a complex language with many data types, the code generator is often allowed direct access to the symbol table constructed by the parser. The information in the symbol table can then be used directly to generate the correct code to access the different data items. The data flow in such a system is illustrated below.



The parser converts the source program into macros, while simultaneously building the symbol table. The code generator then accepts both the macros and the symbol table as input for generating machine instructions.

A macro line consists of a line number, an operation, and that operation's operands: ie. 1 ADD X,Y. In an actual compiler, the line number is usually implicit, and the operation and the operands can be thought of as pointers. The operation is a pointer into a table of macro definitions. The operands are either pointers to the symbol table entries describing the values to be operated upon, or

pointers to previous macro lines indicating the results of previous macro operations.

The paper discusses two particular kinds of macros: computational macros and data reference macros. The following example illustrates both types of macros.

A(I)=B+C(J)*D

```
i       SS      C,J
i+1     MUL     i,D
i+2     ADD     i+1,B
i+3     SS      A,I
i+4     ASSG    i+3,i+2
```

In this example, SS (subscript) is a data reference macro, and MUL and ADD are computational macros.

As an example of computational macro logic, consider integer addition on the IBM 360. The 360 has two Add instructions for integers: 'A' which adds a word of memory to a register, and 'AR' which adds two registers. In generating code for an ADD macro, the code generator must check the location of the values to be added to see if either of the instructions can be emitted directly. If not, the code generator must emit instructions to load one (or both) into registers. If, in the process of finding a register to load into, the code generator must cause the

previous contents of a register to be stored, the new location of the stored value must be recorded. Furthermore, if one of the values to be added is not directly accessable, (ie. a bit string value), the code generator must emit load and shift instructions to isolate that value in a register. Finally, after emitting the appropriate add instruction, the code generator must record the location of the macro's result.

Similar examples of data reference logic are given in Chapter 4.

2.2 INTERNAL TABLES

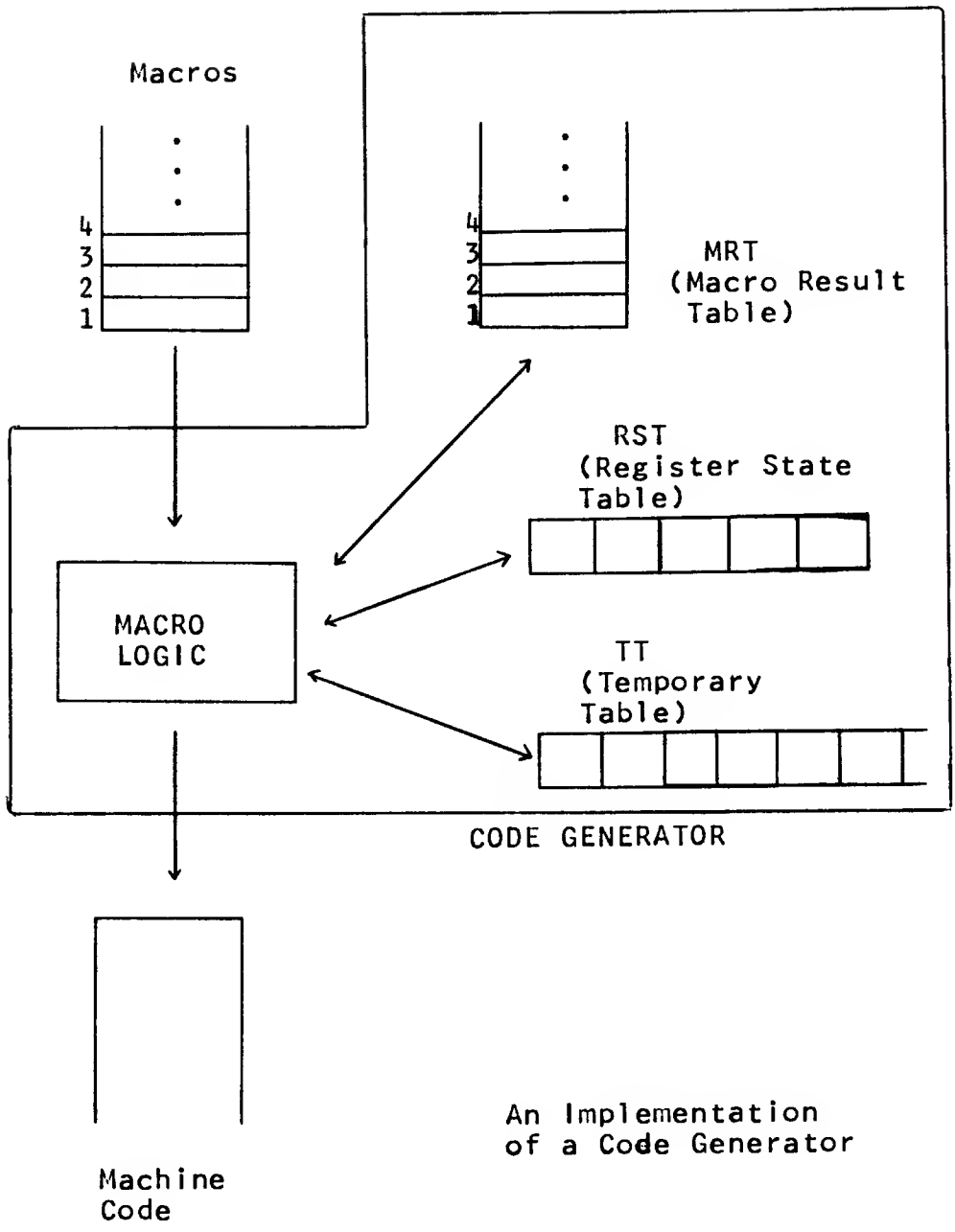
The symbol table contains information about all the values (variables) declared by the programmer. At some point before code generation core locations must be allocated for these variables. The core location information can be stored in the symbol table entry for each item. Exactly how core allocation might be done is discussed in Chapter 4. In addition to the values declared by the programmer, the code generator must also record the location of values which have been computed by previous macro lines, but not yet used. In most machines, a

computation leaves its result in some register. Since the result can often later be used unmoved, it is desirable to leave it in the register if possible. If, however, an intervening macro requires that register for its computations, it is necessary to store its contents in a 'temporary' in core and to remember that this has been done.

To keep track of the location of such previous macro results, three tables are built into the code generator: a macro result table (MRT), a register state table (RST), and a temporary table (TT).

MRT: The macro result table records the location of a macro's result(s) if any. The MRT has one entry for each macro line. Each value recorded in the entry consists of a pointer to the register or temporary where the value is located.

RST: The register state table contains one entry for each register. Each entry indicates whether that register contains a computed value, or if it is free. Each entry recording a computed value contains a pointer to the MRT record representing that value. Thus, when a register must be stored, the MRT entry can be easily



An Implementation
of a Code Generator

changed to point to the temporary location where the value is to be stored. Each RST entry also contains fields which are used to flag a register with information to be used in selecting a register to be stored.

TT: A temporary table can be implemented in various ways. For the purpose of this discussion, any implementation is acceptable. One strategy is to allocate a new temporary each time one is needed, in which case all that need be remembered outside the MRT is the number of the last temporary allocated. A more efficient strategy is to reuse temporaries after the results they hold are used, in which case the TT must have an entry for each temporary allocated.

2.3 THE 'GETREG' ROUTINE

The internal tables described in the preceding section allow computed values to be left in the registers where they are computed. If such tables are not used, every computed value must be immediately stored in a temporary, which is clearly undesirable. If values are to be left in registers, however, a routine must be provided which locates

free registers available for use. The paper refers to that routine as the GETREG routine.

The GETREG routine is passed the name of a register class as an argument. It cycles through that class looking for a free register. If none are found, the routine picks one of the registers and stores its current contents in a temporary, updatings the MRT entry pointing to that value. The priorities used in selecting which register to store, if there is a choice, are discussed in Chapter 3.

2.4 SOME QUESTIONS TO BE ANSWERED

The previous sections give a brief introduction to code generation in general. The remainder of the chapter attempts to use the introduction as a framework within which to outline exactly what aspects of code generation are to be dealt with in Chapters 3 and 4. Among the questions to be clarified are these:

1. What different types of machine structure do the models presented deal with? Clearly there are many different types of machines, ranging from machines like the 7090 with special purpose registers, to machines like the PDP-10 with general purpose registers, to stack machines, and to

microprogrammed machines capable of complicated runtime checks. Similarly, machines have differing addressing mechanisms: byte-addressing, word-addressing, indexed or unindexed, based or not based, directly addressable or paged addressable (as in many small machines), etc. The models presented are not capable of handling all possible machine structures.

2. What kind of values do the models presented deal with? Possible values in a computer are integers of different precision, booleans, bitstrings, floating point numbers of different precision, decimal numbers, character strings, addresses, etc. The present research is not concerned with all of these possible types of values.

3. How are values allowed to map into the machine structure? For instance, are bitstring values to be allowed to cross word boundaries? How are different values assumed to be accessed?

4. What is meant by 'machine description'? Intuitively, one might expect machine description to entail somehow listing registers, core memory units and opcodes. On the other hand, might not a low-level code sequence, which accomplishes some primitive function such as subtraction or

loading a value, be considered to be a reasonable part of a 'machine description'? This question is discussed in Section 2.9.

2.5 ASSUMPTIONS ABOUT MACHINE STRUCTURE

The present research makes several simplifying assumptions about the structure of possible target machines. The assumptions are spelled out in more detail in Chapters 3 and 4.

Registers: The machine is assumed to have a set of registers for manipulating values. These may be either special purpose or general purpose registers. The machine specifier describes the registers by naming them, grouping them into classes, and defining how they are used in manipulating data. Chapter 3 describes more precisely how this is done.

Core Memory: The whole of core memory is assumed to be directly addressable (as opposed to the paged addressability found on some small machines). It is assumed that the addressing is done in a machine instruction by either a displacement and an index, or by a displacement, an index, and a base. The machine specifier must indicate which registers may be used as indices and bases. In generating

an address, DMACS creates an internal '<u>generated address</u>' consisting of a displacement, index, and base (the index or base may be nil). If both index and base are present in a generated address, however, and the particular target machine allows only an index, then DMACS generates code to add the base and index together, thus transforming the 'generated address' into a '<u>machine address</u>' for that target machine.

2.6 ASSUMPTIONS ABOUT VALUES

In a complex real-world compiler, many types of values can be used as operands. Elson and Rake (9) discuss some of the involved problems of writing macro definitions for a complicated language (PL/I). The present work does not attempt to handle the complexity of such a language; rather, it makes certain simplifying assumptions as to the types of values to be allowed as operands. The restrictions allow a reasonably simple model of code generation to be constructed which exposes some of the basic <u>conceptual</u> processes and problems involved, without becoming bogged down in a huge ad-hoc mess.

The model of a code generator presented in this paper is set up to handle values which, intuitively, are of the integer (or integer bit-string) and floating point variety; values which are manipulated via registers and thus are no larger than the registers used on the particular target machine. Character-string and decimal values are not considered.

2.7 HOW VALUES ARE REPRESENTED ON THE MACHINE

There are three general classes of locations for values on a machine: a value can be in a register, it can be simply accessible in core, or it can be in core but not simply accessible. A value is simply accessible if its address can be put directly into a computational machine instruction, such an Add instruction. (Thus a value may be addressable in a special load instruction yet not simply accessible). For instance, a byte on the IBM-360, even though addressable, is not simply accessible for computation. It must first be isolated in a register.

Let us examine how a value might fall into each of these classes.

Registers: The only values which may be in a register (at the start of a macro expansion) are values computed by previous macro lines.

Simply Acccessible: Simply accessible values include both results of previous macro lines which have been stored in temporaries (which are assumed to be simply accessible locations), and values declared in the source program which have been mapped into simply accessible core memory units. Chapter 4 explains exactly how this mapping is done.

Not Simply Accessible: This class is composed of values which cannot be directly operated upon by computational instructions. They must first be isolated in a register before they can be used. Such values include individual bits, and bit-strings which are not on wholely accessable boundaries.

2.8 LOAD/UPDATE ROUTINES

The fact that not all values are simply accessible gives rise to the concept of a load/update pair: a pair of routines to access and to update a value. The idea of characterizing a data item by a pair of load/update routines was first formulated by Strachey (11). A simple example of

such an unaccessible data item is a bit-string within a word. Its location might be represented by an address (perhaps indexed and based) and a bit displacement within the addressed memory unit. Its load/update pair might consist to two routines which take the 'location' and generate code as follows:

1. Load Routine:
    a. load the memory unit (ie., word) into a register
    b. shift left to eliminate high-order bits
    c. shift right eliminating low order bits and right-adjusting the value in the register

2. Update Routine:
    a. shift the new value to the correct target position
    b. load the target word into a register
    c. use a bit mask to zero out the target byte
    d. OR the two words together
    e. store the result

In practice such a value has two kinds of 'location' and correspondingly two load/update pairs: one for when the location of the string within the word is known at compile time, and one for when it is computed at run time. The routines are further complicated if a value extends across a

word boundary.

The load/update problem arises from the fact that programmers are interested in values that do not map directly into accessible units. Generally only an address can be put into a machine instruction. If a computational machine instruction could accept an address, starting bit, and bit length, then the complexity of the load/update routines would disappear. An alternate approach might be to have special hardware load and store instructions to access bits of a word. This would retain the load/update framework, but the routines would consist of only one instruction.

2.9 MACHINE DESCRIPTION

Using DMACS, a machine specifier can implement a language by describing various features of his machine. In the next two chapters, the details of such a description are examined in more detail.

Parts of the 'description' consist of listing names of registers and of core memory units and of describing how these relate to one another. Another part of this description, however, involves writing short low-level code

CHAPTER III

A CODE GENERATOR AS A STATE MACHINE

3.1 OVERVIEW

3.1.1 THE STATE MACHINE

Chapter 3 presents a model of the logic of a computational macro. This model pictures a code generator as a state machine whose state is determined by the location of the values used to generate code. The location of a value may be an accessible core location, a non-accessible core location, or a register. In the model, each computational macro has one or more permitted states for its operands from which code can be emitted. To generate code for a macro, the code generator must make a transition into one of the permitted states and emit a particular code sequence from that state.

In a procedural macro definition language, the user explicitly specifies these transitions himself. In a descriptive system such as DMACS, logic to perform transitions is deduced automatically from machine-descriptive information. The chapter shows how such an automatic mechanism is built into DMACS to perform transitions given a machine description describing register

structure, permitted states for computation, and code sequences which perform these computations. Not surprisingly, the automatic mechanism makes certain restricting assumptions as to object machine structure. Thus, the model is a somewhat restricted one presented to isolate the basic ideas involved, and to provide a basis upon which a more general system can be built.

3.1.2 THE STATE OF THE MACHINE

In this chapter, the term 'state' is used in two contexts: the 'state' of the code generator as a whole, and an input, output, or permitted 'state' of an individual macro.

1. The state of the code generator is determined by the locations of all the values which are to be used as operands to any macro.
2. The input state of a macro is determined by the location of the values passed to it as operands.
3. A permitted state of a macro is a particular configuration of operand locations from which code can be emitted.
4. An output state of a macro is determined by the location of the result of the computation.

3.2 A SIMPLE EXAMPLE

The following simplified example illustrates how the state machine concept is used. The example concentrates on the integer addition for the IBM-360.

1. Input States: For simplicity, let us restrict operands to two locations:

    1. registers of class 'R' (abbreviated 'R')
    2. accessible storage (abbreviated 's')

Thus input states for two operands can be described by the following pairs (s,s), (s,R), (R,s), or (R,R).

2. Permitted States: The IBM-360 has two instructions which perform integer addition. Permitted states are (R,s), (s,R), and (R,R). From (R,s) and (s,R) a storage-to-register Add instruction, 'A', is emitted. From (R,R) a register-register Add instruction, 'AR', is emitted.

3. A Machine Independent Macro: If the source language allowed both integer and floating point operands, the language implementer might write a machine independent ADD macro with logic as follows:

are discussed in Chapter 4. A more detailed description of the 360's register structure is found in Section 3.7.

Next, the machine specifier defines integer addition.

```
IADD a1,a2 (commutative)
from R(a1),R(a2)  emit AR a1,a2     result  R(a1)
from R(a1),S(a2)  emit A  a1,a2     result R(a1)
```

This description defines two permitted states, code to be emitted from each state, and the location of the macro result. In the first state, both operands are in registers. From this state, an 'AR' instruction is to be emitted. The result is to be recorded in the register containing the first operand. The declarations are used to fill out the MIML macro. The attribute 'commutative' indicates that addition is commutative, and thus R(a2),S(a1) will be included as a permitted state without being declared explicitly.

Notice that the declarations are essentially a description of IBM-360 integer addition.

5. Advantages: Because the state machine model is built into DMACS, both the language implementer and the machine specifier find their tasks lightened. The language implementer can write a very simple source macro without worrying about machine structure. He need not perform tests

to ascertain the state of the operands, nor transform the input operand state in any way. The machine specifier, in turn, is able to implement the macros by describing his machine without worrying about the constructs of ths source language or the internals of the compiler.

6. The Role Of DMACS: The machine specifier defines his register structure, the permitted states filling out each 'primitive' (such as IADD) in the machine independent macros, the code sequences to be emitted from each permitted state, and his data pathways including load and store instructions. From this descriptive information, DMACS must deduce three things: how to select a target permitted state for a given input state, how to reach that state, and how to obtain a free register of a given class when, in the process of making a transition, it needs to load a value.

The remainder of this chapter deals with these topics in more detail and discusses the problems involved.

3.3 MACHINE STRUCTURE

3.3.1 REGISTERS

The code generator must be able to manipulate values in and out of registers to attain permitted states. In trying to incorporate automatic register handling logic into DMACS, there are two conflicting goals. First, the user must be able to describe his registers flexibly enough to include a reasonably large class of machines. Second, there must be enough restrictions so that the logic which attains permitted states can be generated from this description automatically. These two goals conflict since the more flexible the model is, the harder it is to incorporate into an automatic system. The assumptions as to register structure outlined in this section are restrictive, but provide a base for later extension of the model.

In attaining permitted states the system must be able to find a free register of a given class, to load and store the contents of any register, and to transfer a value from one register to another. To allow this, the machine specifier defines the following:

1. The Machine Registers: $R = (r1, r2, r3 \ldots rn)$

2. Classes of Registers: (R1,R2,...Rn); $R_i \subset R$
The classes are defined so that every register is in at least one class, if only by itself, and so that any two classes are either subsets, equal, or disjoint. There is no partial overlap.

3. Pathways to Core: Each class of registers is assumed to have a direct path to and from core. There is no need to go through a second register in either loading or storing. This is a simplifying assumption which might be relaxed in a more powerful extension of the model. (A stack machine, for instance, does not conform to this assumption). The machine specifier must define the load and store instructions used in these pathways.

4. Paths between Registers: The machine specifier must define any available register to register transfers.

5. Relationships between Registers: The machine specifier may define relationships between registers. These can be used for such register-register relationships as even-odd pairs. He may also specify that in certain conditions the use of one register implies that a related register must be made available as well.

In this fashion, the user describes his register structure. Section 3.3.3 describes how this information is used by DMACS to construct the GETREG routine to obtain a free register of a given class.

3.3.2 SAMPLE REGISTER DESCRIPTION: IBM-360

```
rclass REG:r2,r3,r4,r5,r6,r7,r8,r9,r10,r11

rclass ODDREG:r3,r5,r7,r9,r11

relation EPAIR (stored:ODDREG)
r3:r2
r5:r4
r7:r6
r9:r8
r11:r10

rpath WORD->REG:  L  REG,WORD
rpath REG->WORD: ST REG,WORD
rpath REG->ODDREG: LR ODDREG,REG
```

These declarations define two register classes. For each member of the class ODDREG, a related EPAIR register is declared. The attribute (stored:ODDREG) means that when an ODDREG register is called for, its related EPAIR register must be made available as well.

3.3.3 THE 'GETREG' ROUTINE

The GETREG routine is called by DMACS when in performing a transition a free register of a given class is needed. The routine must be adjusted by DMACS using the machine specifier's description of his register structure, so that it operates correctly for the particular machine registers and register classes involved.

The GETREG routine cycles through the register class it receives as an argument attempting to find an empty register. If none are empty, the routine must choose a register to store based on the 'flags' attatched to the registers. The flags are used to protect values in registers so that they will not be stored unless necessary. The mechanism for flagging is complicated by the fact that a macro can be called as a subroutine during a given macro expansion. The priorities used in protecting registers are discussed in more detail in Section 3.5. The net effect of the priorities is that the most recently set registers are stored last. The GETREG routine also must handle situations when related registers must be freed at the same time.

3.4 THE AUTOMATIC TRANSITION

3.4.1 INTRODUCTION TO THE TRANSITION

Performing a transition involves transforming any possible input state of a macro into one of that macro's permitted states. The input state of a macro is determined by the location of the values passed to it as operands. These operands can be classed as follows:

1. s - in an accessible storage location in core memory
2. Ri - in a register of register class Ri
3. Rjf - in a non accessible core location, requiring a code generating load function which will isolate the value in a register of class Rj. (The concept of having to apply a function to an operand could apply to mode conversions as well as to loading non-accessible values. Thus, although this paper deals only with Rjf values in a limited context, the concept involved is a more general one.)

For the sake of simplicity, this section deals only with two-operand macros, (such as ADD X,Y). For a two operand macro, input states are taken from

(s U Ri U Rif) X (s U Ri U Rif)

Permitted states are taken from

(s U Rf) X (s U Rf)

On a machine that has one class of registers (say R), input states are (s,s), (s,R), (s,Rf), etc. Permitted states for an arbitrary macro might be (s,R) and (R,R). Thus a reasonable transition to make from input state (R,s) is to permitted state (R,R).

Performing the transition involves choosing a target permitted state to aim for and a path to reach that state. In the remainder of the chapter, we assume that the task of performing a transition can be seen as two distinct problems:

1. Selection of a target permitted state for each input state, based on the cost of transforming each operand location.

2. Given an input state and its target state, determining in what sequence changes are to be made.

These two steps are closely related. On some machines, the two steps can not be performed independently. For instance, on a stack machine one can not consider the cost of transforming the location of each operand individually without considering the sequence of transformations. On a machine that conforms to the assumptions that we have made

about register structure, however, the separation of these two steps is possible.

3.4.2 SELECTING A TARGET STATE

The selection of a target state for each possible input state of a macro is the first step in setting up the automatic transition. If there is only one permitted state, this selection is trivial. Otherwise a target for each input state must be selected from among several permitted states. Clearly, some criteria is needed for measuring the cost of changing states. For each input state, the permitted state yielding the lowest such cost can then be selected as a target. The cost criteria used in this section is the number of instructions required (not counting inadvertent storing of values since these are not predictable in advance). Since we assumed in section 2.3 that each register has a direct path to and from core, the maximum cost of changing the location of one operand is 2 (storing the value from one register, and then loading it into a second). Rif values, which require a function to load them into a register, are treated as if they were already in that register since the function is to be applied in all cases and is thus a constant cost.

Example: Assuming two register classes, R and R', and a register-register transfer, sample costs are:

(input state)	(permitted state)	(cost)	(comment)
(s,s)	(s,R)	1	load
(s,R)	(s,R)	0	
(s,R)	(R,s)	2	store, load
(s,R)	(s,R')	1	transfer
(s,Rf)	(R,R)	1	load
(Rf,Rf)	(R,R)	0	

Figure 3.1: Sample Transition Costs

An alternate cost criteria might be instruction execution times. In either case, the target selection algorithm simply selects for each input state that permitted state which can be reached at lowest cost. The selection of a target state for each input state need not be performed every time the macro is called. It can be compiled into a table at machine description time.

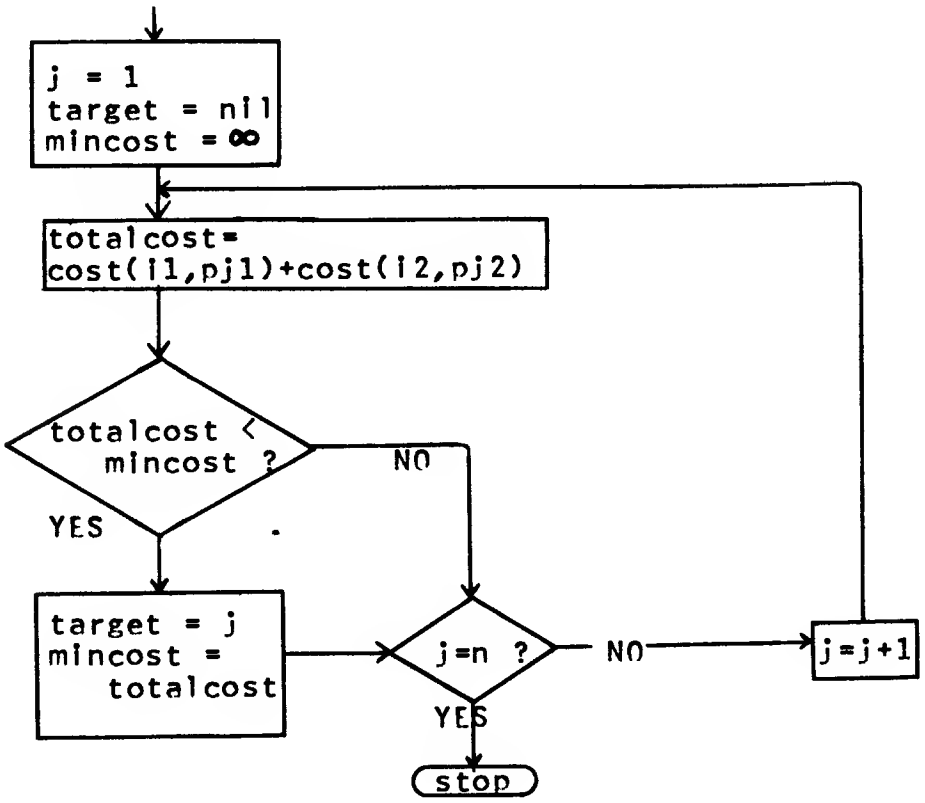
3.4.3 SEQUENCING THE TRANSITION

Once a target state has been selected for each input state, there remains the problem of deciding the order in which changes are to be made. This chapter first outlines a general strategy which accomplishes sequencing for all possible machine structures. The general strategy is called 'blind' sequencing for reasons that become apparent. The

Algorithm to select a target permitted state
for an input state of a macro

Input state: i = (i1,i2)

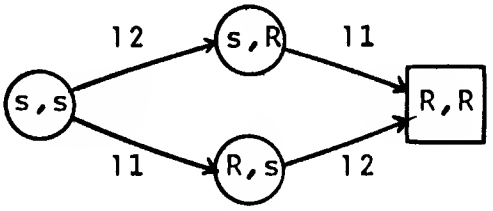
Permitted states: pj = (pj1,pj2) j = 1,n



The function cost(i,p) determines the number of instructions required to change i to p.

chapter later discusses a second strategy called 'predictive' sequencing which, when possible, might be more efficient. The following discussion concentrates on 'blind' sequencing, since blind sequencing always works and is a good vehicle for outlining the problem involved.

To illustrate the problem, let us consider the transition from input state (s,s) to target state (R,R). There are two possible paths, as this graph indicates:



In the graph, each node represents a state, square nodes represent permitted states, and paths from one state to another are represented by arcs. The arcs are labeled to indicate what operation is being performed to which operand. Thus 11 indicates that the first operand is being loaded when that arc is followed.

In the graph, there are two paths from (s,s) to (R,R). In this particular example, the paths are equally efficient and either can be selected. An important thing to notice in this example is that every arc connects one

possible input state to another. This is true because we have taken all possible combinations of locations as possible input states. Since each arc must lead to another possible input state, each state can be examined in turn and one labeled arc can be drawn from its node. Drawing this one arc for each state completes the graph. A decision procedure for determining which arc to draw for a given state is given in Figure 3.2.

3.4.4 ACCIDENTAL TRANSITIONS

There is one complication to be considered in performing the sequencing. It is illustrated below:

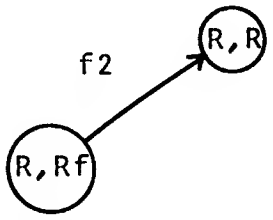


Figure 3.2: Sample Sequencing Graph

In input state (R,Rf), the first operand is already in a register. A code generating function is to be applied to load the second operand. The function can generate code using registers, and even might call macros as subroutines to perform runtime computation. For instance, to load a bit

Algorithm For Blind Sequencing

Problem: Given an input state and a target state, determine what operand to transform first. (Making this decision for each possible input state completes the graph).

Each operand can be expressed as one of the following:

Input	Target
s	s
R	R
Rf	R' (R' ≠ R)

The first operation to be performed to a given operand can be expressed as follows:

Input	Target s	R	R'
s	nil	l	l
R	st	nil	t
Rf	f1	f2	f3

l='load'
st='store'
t='transfer'
fi='apply function f'

These operations can be given priorities:

st > f1 > f3 > f2 > t > l

The effect of the priorities is to give highest precedence to storing values, next highest to applying functions, then to register transfers, and finally to simple loads.

Sequencing is done by labelling each operand of each input state by st, f1, f2, f3, t, l; and then drawing the arc corresponding to the operation with the highest priority. If both have equal priority, then either can be picked.

Figure 3.3

value extending across a word boundary it is necessary on the 360 to load a register pair and perform double word shifts to isolate the value. Thus, the function may require the register containing the first operand to be stored. If this happens, a transition to (s,R) occurs, rather than to (R,R). This is called an accidental transition from an unstable state to an alternate state. To accomodate such unexpected but unavoidable transitions, the graph is augmented to include dotted arcs from such unstable states to the appropriate alternate state.

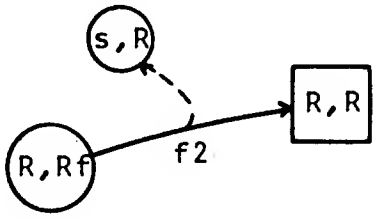


Figure 3.4: An 'Unstable' State

The graph in Figure 3.4 indicates that applying f2 to the input state (R,Rf) should lead to (R,R) but might lead to (s,R). Full examples of such graphs are given in Figures 3.5 and 3.6.

Let us consider for a moment how such accidental transitions can be implemented in DMACS. In the performance of the transition (R,Rf)->(R,R), the register containing the

input states: (s U R U Rf) X (s U R U Rf)
permitted states: (R,s), (R,R)

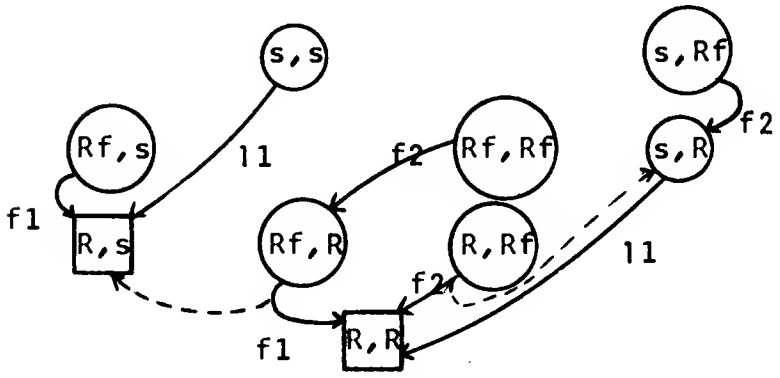


Figure 3.5

input states: (s U R U R' U Rf) X (s U R U R' U Rf)
permitted states: (R',s), (R',R) $R \cap R' = \emptyset$

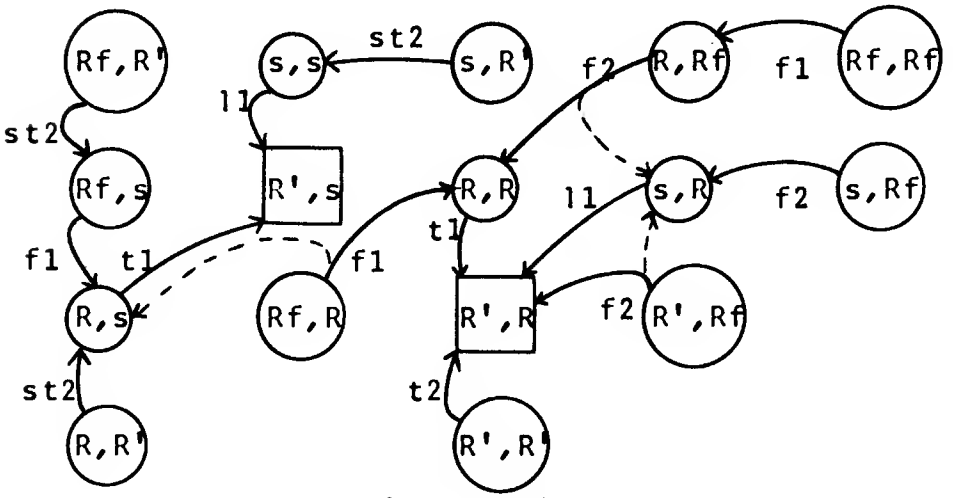


Figure 3.6

Blind Sequencing Graphs

first operand should not be stored unless necessary. This can be assured by flagging that register's entry in the RST (register state table). When storing any register, the GETREG routine must attempt to respect these register flags. After the function has been completed, the code generator must check to see if the first operand is still in a register. If not it must follow the dotted arc to reach its next state.

3.4.5 A REVIEW OF SEQUENCING

The sequencing strategy described in the previous sections is called 'blind' sequencing because it involves applying load functions without looking ahead to see how these functions behave. Consider the following situation:

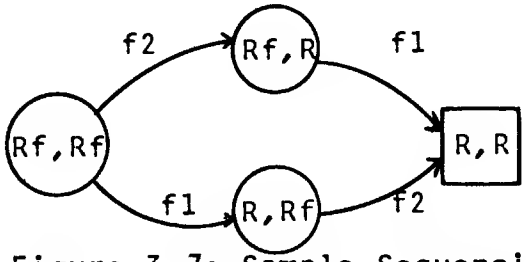


Figure 3.7: Sample Sequencing Graph

'Blind' sequencing arbitrarily selects one of these paths. If it is possible to determine in advance whether a given function can be applied without disturbing a value already

in a register, then a more effective strategy can be used. The second strategy is called 'predictive' sequencing. Predictive sequencing is not so general as blind sequencing since for an arbitrary machine with register-register relations like even-odd pairs, the exact needs of an arbitrary function (even if quite simple) may be very difficult to predict and control.

This chapter concentrates on the blind sequencing approach. Predictive sequencing is mentioned primarily to put the problem in perspective. The main argument against developing a general predictive sequencing strategy for an arbitrary machine is that it would be very difficult to design, and would run the risk of using more instructions at compile time than were ever saved at runtime. In code generation, it is generally true that if elaborate optimization is to be done, it is most profitably done on a fairly global basis, such as allocating registers over loops, removing invariance from loops, consolidating common subexpressions, etc. Blind sequencing has the advantage of affording a degree of local optimization (compared to a system which stores all registers before calling a function) without any really elaborate machinery.

This concludes the introduction to sequencing. Let us step back for a moment and evaluate briefly what these blind sequencing graphs imply. A blind sequencing graph is compiled at machine description time by DMACS. Any particular graph applies to a particular machine, but the concept of such a graph is a general formalism and is applied to all machines.

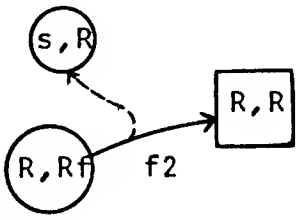


Figure 3.8: Sample Sequencing Graph

To understand the significance of this fact, let us consider what factors govern how frequently the accidental transition is followed from (R,Rf) in Figure 3.8. The frequency depends both on the source program and on the target machine. If the source program uses data-types which require simple accessing functions, then the dotted arc will tend to be followed less often than if more complex data-types are used. Similarly, if the machine has many available registers, the dotted arc will tend to be followed less often than if the machine has few of them.

Notice, however, that neither the programmer nor the machine specifier need even know that the problem exists. For that matter, neither need the language implementer. Only one person need ever worry about it- the DMACS designer who does all the worrying for everyone.

3.4.6 A GENERAL OVERVIEW OF A TRANSITION

The automatic transition mechanism looks at the type of a macro's operands, looks at the permitted states, and then initiates one or more transformations to attain one. This process can be described in general terms:

1. A system is in a given state (certain values are in certain locations).
2. It is desired to transform the system to a new state with certain properties (particular values in particular locations).
3. Functions are available which can effect a desired local change to part of the system, but with possibly unpredictable side-effects. (A load function applied to an Rif value might store an arbitrary value from a register).
4. It is desired to make a sequence of such local changes and still have the resulting global state

well-defined.

To accomplish this goal, the mechanism that generates permitted states must be able to detect when one function it applies stores a register that it expected to be loaded, and either reload that value or else pick an alternate permitted state. Two potential problems in a system of this sort are deadlock and thrashing.

1. Deadlock: Deadlock occurs if a value is irrevocably locked into a register by the flagging mechanism, so that it can not be stored. If this were possible, it is easy to imagine a situation where a macro called as a subroutine might be unable to obtain the registers it required. Such a deadlock can not occur in the system outlined here, since the register flags are only interpreted as requests that a register not be stored unless necessary.

2. Thrashing: In section 3.4.4, accidental transitions were described. In such transitions, an attempt to reach one state results in an inadvertent transition to an alternate state. One might wonder whether such inadvertent transitions could be repeated indefinitely. If so, then a thrashing situation might result, in which each sucessive attempt to reach a permitted state is thwarted. Fortunately,

of a load function may result in macros being called as subroutines, each macro invocation is numbered sequentially. That number is used to flag registers. In obtaining a register, the GETREG routine uses the following priorities:

1. an empty register
2. an unflagged register
3. a register with the lowest flag (ie least recently set)

Thus the most recently computed values are the most securely protected. As a result, if a register is loaded and no arbitrary functions are called it can be relied upon to remain in its register.

The process of macro expansion involves performing the following steps, I, II, and III in sequence:

I. Protect Values Already in Registers: First any values to be used by the macro which are already in the correct registers are flagged. Such values include operand values as well as values to be used as indices or bases to obtain an operand value. If a value in a register requires that a related register be stored, then make sure that register is stored and flag it as well.

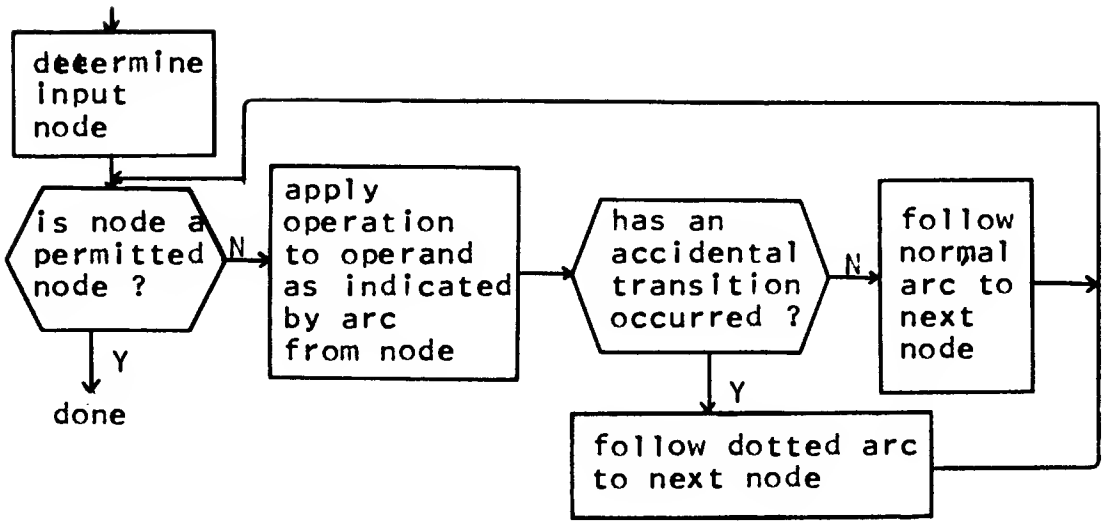


Figure 3.9: Performing a Transition

II. Perform the transition to a Permitted State: Notice that it is in the process of following the sequencing arcs that the load functions and the GETREG routine are called as subroutines. Load functions are called when a load is applied to an Rif value. The GETREG routine is called when a load or transfer arc is traversed.

III. Perform Emission and Bookkeeping:

1. For each operand in storage, load any index or base values which are not already loaded.
2. Erase all RST flags set by this macro.
3. Emit the code sequence associated with the permitted

node attained using the sequencing graph. The code sequence was specified by the machine specifier when defining permitted states.

4. Erase the operands from the MRT and RST
5. Record macro result, if any, in the RST and MRT

3.6 AN EXTENSION: OPERATIONS TO MEMORY

A useful extension to the state machine concept, as outlined, is to incorporate 'operation-to-memory' instructions, such as 'add-to-storage'. It is simple to include this common class of instructions by allowing the user to specify alternate destinations for a macro result.

Example: For the PDP-10, which has such instructions, an OMML definition for IADD (defined in section 2.1.2) can be:

```
IADD a1,a2
from REG(a1),REG(a2)  emit IADD a1,a2   result  REG(a1)
from REG(a1),WORD(a2) emit IADD a1,a2   result  REG(a1)
                or    emit IADDM a1,a2  result WORD(a2)
```

(The IADD and IADDM instructions being emitted are PDP-10 opcodes.) The second state declaration indicates that if a1 is in a register, and a2 is in core, then an IADD instruction yields a result in the register, and an IADDM instruction yields a result in core.

To take advantage of such information, only four modifications need be made to the logic outlined in this chapter:

1. When emitting code: If such a choice exists and the core location is a temporary, then defer emitting the instruction, and flag the Register in the RST, indicating the two instructions and the core location.

2. In the GETREG logic: emit an operation-to-memory instruction in preference to explicitly storing a value.

3. In selecting a permitted target state: If there is a choice of input states due to deferral of such an instruction, then evaluate both possible input states and select that one whose target has least cost. If the selection requires emission of an operation-to-register instruction, continue to defer its emission until it is clear that the value need not be stored.

4. After sequencing and prior to code emission: First emit any necessary op-to-register instructions for input operands which have been defered.

Although these simple modifications to the DMACS logic certainly lead to no dramatic gains in efficiency, they do represent a useful extension to the state machine concept.

3.7 SAMPLE MACHINE DESCRIPTIONS

This section outlines the logic of two simple machine independent macros which might be written in MIML. Then OMML descriptions of the IBM-360 and of the PDP-10 which fill out the macros are given.

Machine Independent Macro Logic:

```
macro MUL X,Y
          if the types of X and Y are integer
                    then IMUL X,Y
          else if the types of X and Y are floating
                    then FMUL X,Y
          else error

macro SUB X,Y
          if the types of X and Y are integer
                    the ISUB X,Y
          else if the types of X and Y are floating
                    then FSUB X,Y
          else error
```

OMML Machine Description of the IBM-360:

The IBM-360 has one set of registers for integer arithmetic and another set for floating point arithmetic, and therefore has separate pathways to and from these registers. For multiplication and division of integer operands, even-odd pairs of registers are used.

```
rclass REG:r2,r3,r4,r5,r6,r7,r8,r9,r10,r11
rclass ODDREG:r3,r5,r7,r9,r11
rclass FREG:fr0,fr2,fr4,fr6
relation EPAIR (stored:ODDREG)
          r3:r2,r5:r4,r7:r6,r9:r8,r11:10

rpath WORD->REG: L REG,WORD
rpath REG->WORD: ST REG,WORD
rpath FREG->WORD: LE FREG,WORD
rpath WORD->FREG: STE FREG,WORD

IMUL m1,m2 (commutative)
from ODDREG(m1),REG(m2) emit MR EPAIR(m1),m2  result ODDREG(m1)
from ODDREG(m1),WORD(m2) emit M EPAIR(m1),m2  result ODDREG(m1)
```

(On the IBM-360, multiplication requires one operand in an 'odd' register. The multiply instruction must refer to its even pair.)

```
ISUB s1,s2
from REG(s1),REG(s2) emit SR s1,s2  result REG(s1)
from REG(s1),WORD(s2) emit S s1,s2  result REG(s2)

FMUL m1,m2 (commutative)
from FREG(m1),FREG(m2) emit MER m1,m2  result  FREG(m1)
from FREG(m1),WORD(m2) emit ME m1,m2 result FREG(m1)

FSUB s1,s2
from FREG(s1),FREG(s2) emit SER s1,s2  result FREG(s1)
from FREG(s1),WORD(s2) emit SE s1,s2  result FREG(s1)
from FREG(s2),WORD(s1) emit LNER s2,s2;AE s2,s1  result FREG(s2)
```

(Notice that since a 'complement register' instruction, LNER, exists for floating point, a state can be specified with s2 in a register and s1 in core).

OMML Machine Description of the PDP-10:

The PDP-10 has one set of registers for both integer and floating point arithmetic. Since the PDP-10 has operation-to-memory instructions, all memory-register state declarations include two alternate destinations.

```
rclass REG:a,b,c,d,e,f,g,h,i,j,k,l,m,n
rpath REG->WORD: MOVEM REG,WORD
```

```
rpath WORD->REG: MOVE REG,WORD


IMUL m1,m2 (commutative)
from REG(m1),REG(m2) emit IMUL m1,m2  result REG(m1)
from REG(m1),WORD(m2) emit IMUL m1,m2  result REG(m1)
                     or  emit IMULM m1,m2  result WORD(m2)


ISUB s1,s2
from REG(s1),REG(s2) emit ISUB s1,s2  result REG(s1)
from REG(s1),WORD(s2)  emit ISUB s1,s2  result REG(s1)
                     or emit ISUBM s1,s2  result WORD(s2)

FMUL m1,m2 (commutative)
from REG(m1),REG(m2) emit FMPR m1,m2  result REG(m1)
from REG(m1),WORD(m2) emit FMPR m1,m2 result REG(m1)
                     or emit FMPRM m1,m2 result WORD(m2)

FSUB s1,s2
from REG(s1),REG(s2) emit FSBR m1,m2 result REG(s1)
from REG(s1),WORD(s2) emit FSBR m1,m2  result REG(m1)
                     or emit FSBRM m1,m2  result WORD(m2)
```

3.8 SUMMARY: THE STATE MACHINE

The chapter outlines how a code generator performing computations can be pictured as a state machine. Then it shows how the state machine can be formalized and incorporated into DMACS, a system for building machine independent code generators.

Once the state machine model is incorporated into DMACS, it becomes a tool that a language implementer can use. It is a convenient tool since it frees the language implementer from worrying about machine structure, from

having to perform tests to determine input states to his macros, and from having to implement transitions to permitted states. Thus, the macro logic that the language implementer specifies need only deal with particular semantic features of his source language. Therefore the semantics of the source language are logically divorced from any one target machine's structure. As a result, these macros become much simpler to write. Also, once these machine independent macros are written, they can be implemented for a variety of machines from a machine description.

CHAPTER IV

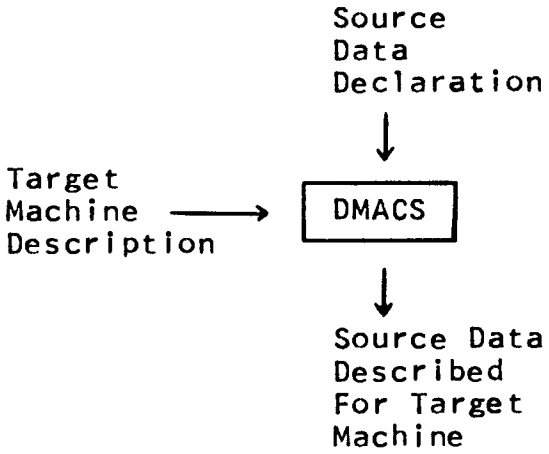
DATA REFERENCE MACROS

4.1 INTRODUCTION

4.1.1 OVERVIEW

Chapter 3 described a state machine model which is built into DMACS and used as a tool to create machine independent macros which can be filled out from a machine description. The state machine is useful to help model computational macros.

Chapter 4 turns to the problem of achieving the same machine independence for data reference macros. To achieve this goal, a data definition facility is built into DMACS.



A machine specifier describes his machine memory structure and describes how source data items are mapped into that memory. From this description DMACS characterizes source data items in terms of the primitives of the data definition facility. The language designer writes his data reference logic in terms of the primitives of the facility using two built-in functions, called the INCREMENT and CONVERT functions. These functions operate on the primitives of the data definition facility. In effect, these two functions represent a machine independent model of data reference logic. A language implementer can write data reference macros in terms of these built-in functions without worrying about how the data items of his language map into the core memory of a particular machine.

Chapter 4 is not an extension of Chapter 3. It pursues a similar goal in a new area: machine independence for data reference macros similar to that acheived in Chapter 3 for computational macros.

4.1.2 DATA REFERENCE

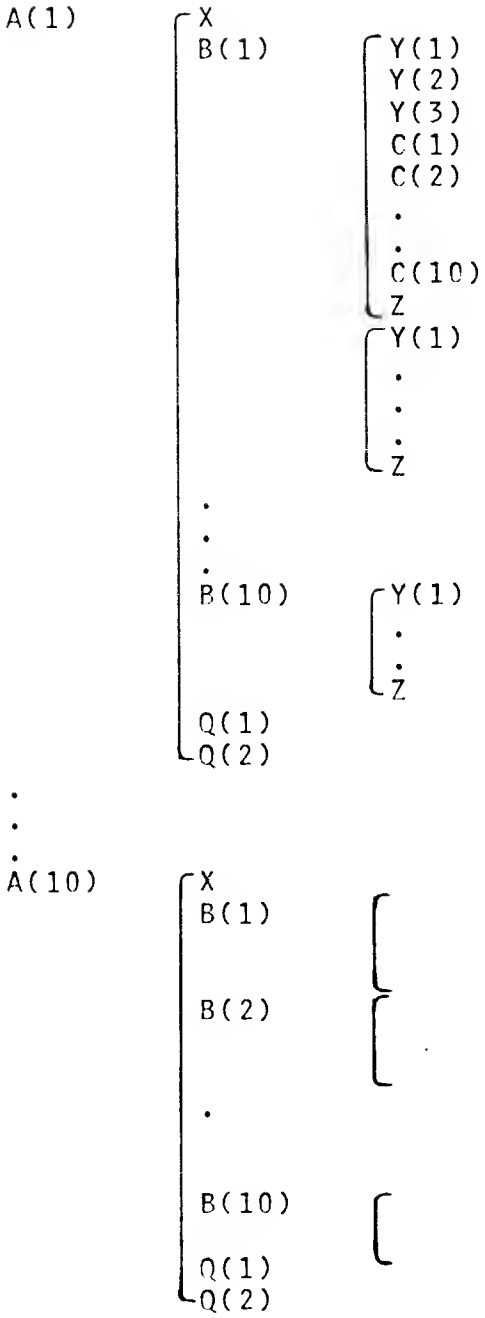
This section introduces the reader to the term 'data reference' as used in this chapter, and gives a simple example of data reference macros in action. The data

reference constructs dealt with in this chapter are subscripted structures as found in PL/I. Pl/I is chosen both because it is a well known language and also because it has powerful data referencing constructs. For simplicity the chapter deals only with structures whose size is static and known at compile time. This restriction eliminates some of the messiness of PL/I's structure implementation and lets us concentrate on the basic problems of making such references machine independent. If we allow dynamically varying structure sizes, then we must worry about what logic can be performed at compile time and what logic must be performed at run time by generated code. Restricting our attention to static structures frees us to concentrate more fully and more clearly on machine independence of data reference, rather than on the details of implementing dynamic structures for PL/1. The restrictions still allow useful and flexible data referencing constructs.

A sample structure is the following:

```
declare 1 A (10) fixed,
        2  X,
        2 B (10),
                  3 Y (3),
                  3 C (10),
                  3 Z,
        2 Q (2);
```

This declaration defines a subscripted structure. The



Sample Structure Layout

```
I SS      SS        A,I
I+1       SUBST     I,B
I+2       SS        I+2,J
I+3       SUBST     I+2,C
I+4       SS        I+3,K
```

This approach can handle any structure reference in a simple, general fashion.

Having discussed the data reference macros to be dealt with, we now present a simple example of how code might be generated for the macros outlined above. This simple example assumes that the items all represent full words of data on some particular machine. Later we shall extend this simplified situation to allow more complicated data items.

Each structure item is characterized by two numbers:

1. an offset from the beginning of its substructure element
2. an element length

The structure item B, for instance, has an offset of 1, and an element length of 14.

In generating code for the macros above, two running totals can be kept: compile time words- CW, and runtime words- RW. The running total represents a displacement into the structure. At the end of the set of macros, the

displacement points to the correct terminal data item. The following logic illustrates how the macros might be expanded. In the interest of clarity and simplicity, the code generated is not as optimal as it might be.

> SS A,I records the offset of A, which is 0, in CW, and generates code to multiply the element length of A (143) by I-1. The result of the multiplication becomes RW. (I-1 is used in the multiplication on the assumption that the first (zeroth) element is defined as A(1).)
>
> SUBST I,B adds at compile time the offset of B, which is 1, to CW.
>
> SS I+1,J generates code to multiply the element length of B, 14, by J-1 and add the result to RW.
>
> SUBST I+2,C adds at compile time the offset of C, 3, to CW.
>
> SS I+3,K generates code to add K-1 to RW. (no multiplication is necessary since the element length of C is 1).

The result of all the computation is a pair of values (CW, RW) which represent a compile time displacement and a runtime index pointing to the desired data item. On a machine like the IBM-360, this pair can be put directly into a machine instruction (ie. Load, Add) to access that data item.

The above example illustrates the general operation of data reference macros. It is shown later that an expanded, but similarly clean, framework can be used to handle data

more complicated than the full word items of this example. When data items are bytes and bitstrings, the logic of the macros is somewhat more complicated, and the eventual result is not a simple full word pointer, but rather a 'location' that can be input to a load/update routine which accesses the data-item pointed to.

The most important point to notice in the example is that each structure item is characterized by an offset and an element length and that on different machines, these offsets and lengths might be different. A terminal data item is also characterized by two additional parameters, a load/update pair to access the item, and a data length which need not be the same as the element length (For instance, an array of 5-bit bitstrings alligned on word boundaries would have a data length of 5 bits, but an element length of one word.) These too could vary for different machines.

4.2 THE DATA DEFINITION FACILITY

4.2.1 DESCRIPTION OF DATA

The previous section examined a simple example of data reference. This section presents a more precise framework for describing the type of data with which the chapter is concerned. A data item can be characterized by a 4-tuple

(OF,EL,DL,LU) describing how it is implemented on a particular machine.

OF- the offset of the data item from the origin of the structure element to which it belongs

EL- the element length of the structure element which that data item defines

DL- the data length- the length of the piece of data which the data item represents

LU- the load/update pair which accesses the data item.

OF and EL can characterize any data item. DL and LU apply only to terminal data items.

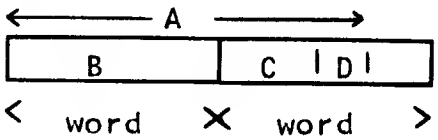
The following example illustrates how data items declared in a particular source program might be implemented differently on two different machines, the IBM-360 and the PDP-10:

```
declare 1 A packed,
        2 B fixed,
        2 C char (2),
        2 D char;
```

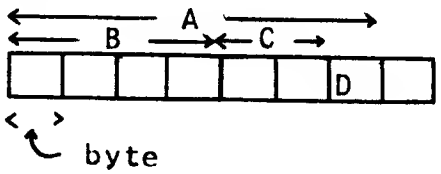
The PDP-10 is a word addressed machine with 36 bits/word. Assume a data item of type 'fixed' to be defined as a word item, and a character to be defined as a nine bit item. The IBM-360 is a byte addressed machine with 8 bits/byte. Assume a fixed data item to be defined as a word (four byte) item, and a character to be defined as a byte. Section

4.2.3 describes how such definition is done. Granting these assumptions, the storage layout for structure A is as follows:

PDP- 10:



IBM- 360:



Thus the data item 'A.D' is described as follows on the two machines:

1. on the PDP-10

OF- 1 word, 18 bits
EL- 9 bits
DL- 9 bits
LU- the loadupdate routine for bitstrings

(Notice, as an aside, that if one wanted to pack 5 seven-bit characters into a word, then instead of an element length, this item would have two numbers associated with it, 36 and 5, Any index into an array of such characters would be multiplied by 36 and divided by 5 to yield a bit displacement.)

2. on the IBM-360

OF- 6 bytes
EL- 1 byte
DL- 1 byte
LU- the load update routine for bytes.

4.2.2 DATA DEFINITION

To allow data reference macros to operate over data which can be described differently for different machines, certain problems must be solved.

1. Suitable primitives must be found, flexible enough to describe offsets and lengths of data for a number of machines.

2. An algorithm must be written which takes a structure declaration and a machine description and computes offsets and lengths describing that data for that machine, expressed in terms of these primitives.

3. Data reference macros must be written in terms of these primitives, so that these macros will be machine independent.

DMACS solves these three problems by using a built in data definition facility. The primitives of the data definition facility are addressable units and bits. All data is ultimately described in these terms.

The remainder of the chapter first outlines how these primitives can be deduced from information supplied by a

machine specifier. The chapter then illustrates how DMACS can help a language implementer write data reference macros in terms of these primitives.

4.2.3 DEDUCTION OF PRIMITIVES FROM A MACHINE DESCRIPTION

DMACS characterizes data for any machine in terms of addressable units and bits. Information to make this characterization must be deduced from the machine description which specifies the following:

1. core memory units: The machine specifier defines his core memory units (such as bits, bytes, words, double words, etc.), how these map into each other, and which is addressable.

A sample declaration for the IBM 360 follows:

```
mem BIT
mem BYTE (8 BIT, addressable)
mem WORD (4 BYTE, boundary 4)
mem DWORD (8 BYTE, boundary 8)
```

The attribute 'boundary 4' indicates that an element with storage class WORD has an address congruent to zero, modulo 4.

2. Source data types: The machine specifier must indicate which storage unit each source data type is to be mapped into. It is here that a character data item might be defined

as a byte on the IBM-360 and as a bitstring on the PDP-10. For a language where data can be packed or aligned, both storage units are indicated. DMACS uses this information together with the core memory unit information to determine the offsets and lengths of data items from a source program.

```
map fixed to WORD
map char to BYTE
map bit unalligned to BIT
map bit alligned to BIT allign WORD
```

The last declaration indicates that when a 'bit' data item has been declared to be 'alligned', it is to be alligned on a WORD boundary.

3. Load/Update routines: For each of the memory units, the machine specifier must define a load/update routine to access source data items mapped by the specifier into that memory unit.

Some storage classes may bhave simple routines:

```
mpath WORD->REG: L REG,WORD
mpath REG->WORD: ST REG,WORD
mpath BYTE->REG: SR REG,REG;IC REG,BYTE
     (etc.)
```

Other load/update routines are more complicated, and are discussed in section 4.2.4.

When a language data declaration is processed, information from such a machine description must be used to compute offsets and lengths for each data item. The offsets and lengths can each be described by a 2-tuple (addressable units, bits). The tuple (4,2), for instance, stands for 4 addressable units and 2 bits.

As a simple example, consider the following structure:

```
declare 1 Z,
        2 A fixed,
        2 B bit (12),
        2 C bit (3),
        2 D 2 bit alligned;
```

The following table indicates how the structure might be described for the IBM-360 and the PDP-10:

(data)	(offset)(length)		(offset)(length)	
A	(0,0)	(4,0)	(0,0)	(1,0)
B	(4,0)	(0,12)	(1,0)	(0,12)
C	(5,4)	(0,3)	(1,12)	(0,3)
D	(8,0)	(0,2)	(2,0)	(0,2)
	IBM-360		PDP-10	

Each tuple represents addressable units and bits. A is a fixed data item which is mapped into a full word on both machines (and hence 4 addressable units on the 360), and B, C, and D are mapped into bits. The flowchart of a general algorithm which will take a structure and describe it using these primitives is given in Figure 4.1.

This algorithm illustrates how offsets and element lengths can be computed for data items. Two stacks are used: DISP and STACK.

The stacks are pushed each time a new structure level is encountered, and are popped each time a level ends. Each entry of DISP has two fields, one for addressable units and one for bits, which record displacement from the beginning of the current structure level. STACK is used to store the name of the current data item at each level. For each data item an offset (OB) and an element length (EL) is computed.

4.2.4 COMPLEX LOAD/UPDATE ROUTINES

The previous section gave examples of simple load/update routines for addressable data items. Load/update routines for non-addressable items (ie. bit strings) are more complicated for several reasons.

1. They take as input an address, a bit displacement, and a bit length.
2. Bit displacements can be runtime or compile time values.

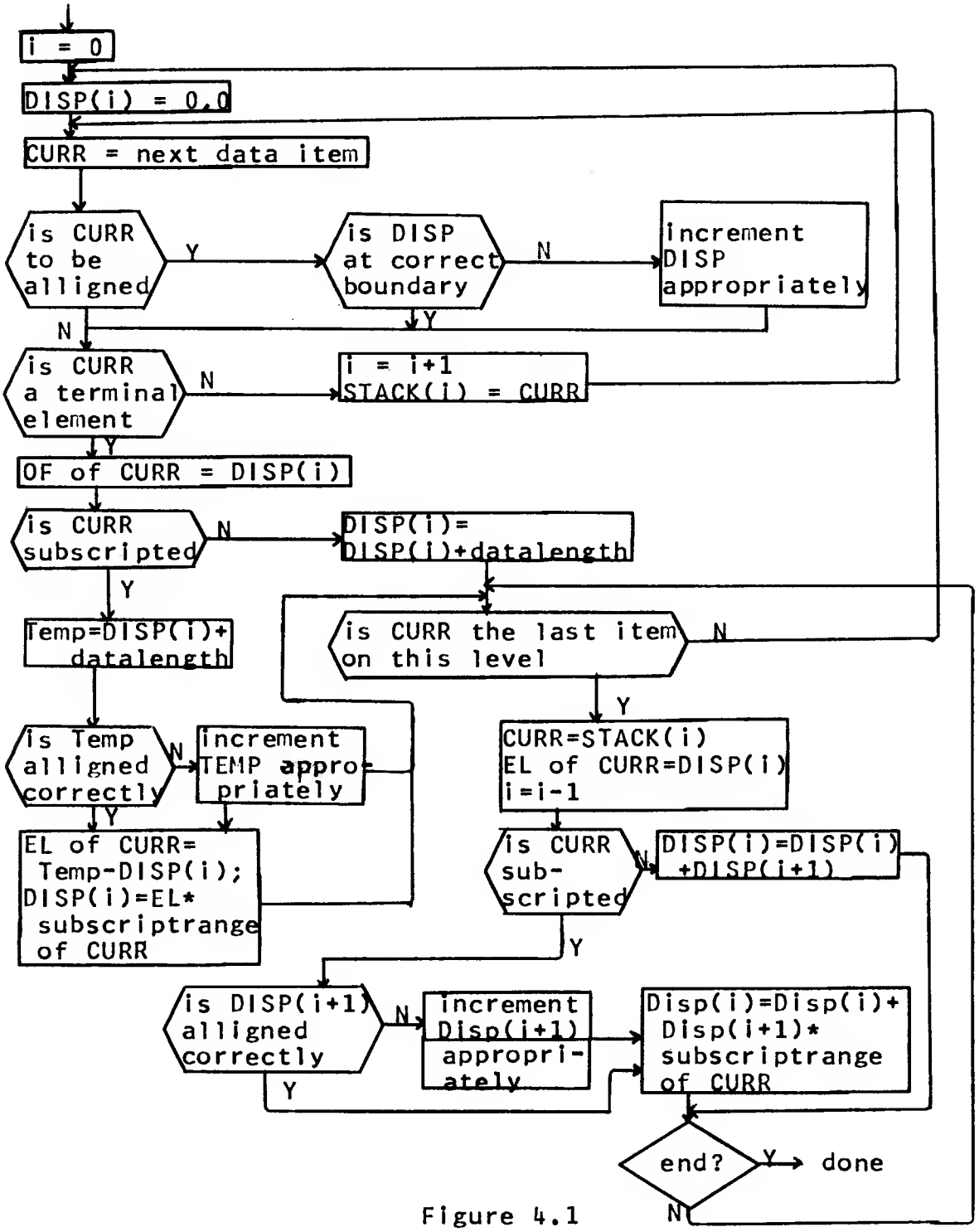

i = 0
DISP(i) = 0.0
CURR = next data item
is CURR to be alligned
Y
is DISP at correct boundary
N
increment DISP appropriately
Y
N
is CURR a terminal element
N
i = i+1
STACK(i) = CURR
Y
OF of CURR = DISP(i)
is CURR subscripted
N
DISP(i)= DISP(i)+datalength
Y
Temp=DISP(i)+ datalength
is CURR the last item on this level
N
Y
is Temp alligned correctly
N
increment TEMP appro- priately
Y
EL of CURR= Temp-DISP(i); DISP(i)=EL* subscriptrange of CURR
CURR=STACK(i) EL of CURR=DISP(i) i=i-1
is CURR sub- scripted
N
DISP(i)=DISP(i) +DISP(i+1)
Y
is DISP(i+1) alligned correctly
N
increment Disp(i+1) appropri- ately
Disp(i)=Disp(i)+ Disp(i+1)* subscriptrange of CURR
Y
end?
Y
done
N

Figure 4.1

3. Bitstrings can run across word boundaries.

Two possible solutions are available to handle this problem. The easiest solution is to require that the machine specifier provide a subroutine which makes the appropriate checks, and executes the correct load and shift instructions for the different situations. The second solution is to allow the machine specifier to define open code sequences to be generated, at least for the simpler cases (for instance, when bit displacement is known at compile time, and hence it can be determined that the item does not cross a word boundary).

A sample load routine for the IBM 360 might appear somewhat as follows:

```
mpath BIT->REG:     L REG,WORD
                    SLL REG,DISP
                    SRL REG,32-LEN
```

The whole problem of exactly how to allow a machine specifier to define open sequences of this sort is a difficult one. It is to a large degree an implementation problem for a DMACS builder, rather than a conceptual problem of machine independence. It is therefore left somewhat open in this chapter.

4.3 MACHINE INDEPENDENT MACROS

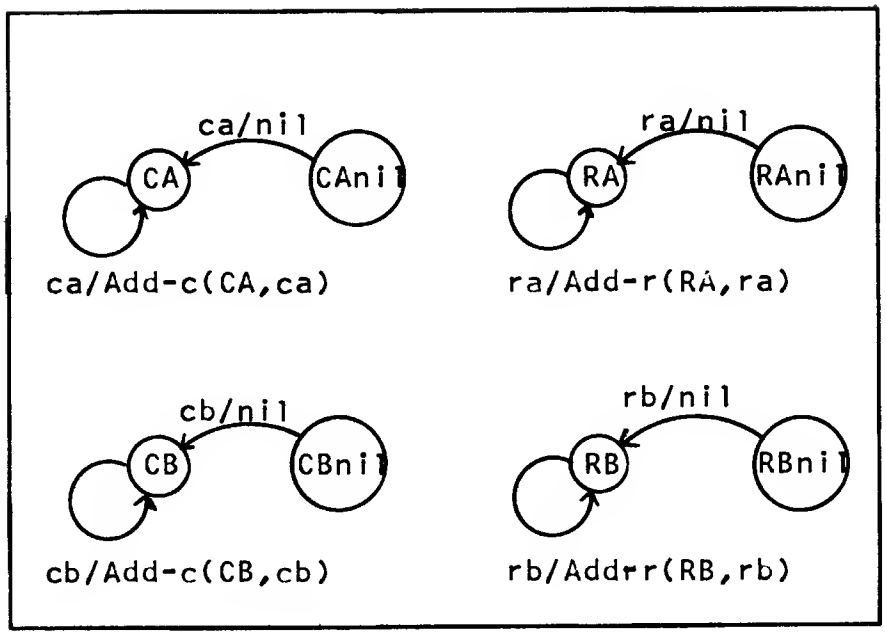
4.3.1 DATA MACRO LOGIC

The previous sections have illustrated how data items on any machine can be characterized in terms of the primitives of a data definition facility. This section describes how machine independent data macros can be written in terms of the primitives of the facility in a clean, simple fashion.

As outlined in section 4.1.2, the operation of a data macro consists of incrementing a pointer into a data base, a pointer consisting of both runtime and compile time values. In the machine independent macros which a language implementer writes, all offsets and lengths are expressed in addressable units and bits. Thus the pointer being incremented can be seen as a 4-tuple: (CA,RA,CB,RB).

CA- compile time addressable units
RA- run time addressable units
CB- compile time bits
RB- run time bits

Any element may be nil: ie. if CA is nil, the pointer has not been incremented by any compile time addressable units. The process of incrementing the pointer can be expressed by the following graph, called the INCREMENT function:



This graph contains four pairs of states. Each pair is a state machine recording the presence or absence of one element of the 4-tuple. Hence, if the pointer has no runtime bits, the third pair is in the state 'nil'. Each pair starts in the state 'nil'. Input is represented by 'ca', 'cb', 'ra', and 'rb'. Actions to be taken are either Add-c, representing addition at compile time, or Add-r, representing addition at run time, or nil. When the pointer is incremented by a new value, the appropriate state machine makes a transition. If this is the first transition for that machine, then there is a change of state with no action performed. If this is not the first transition, then either a compile time 'Add-c' is performed, or code is generated to

to perform a run time 'Add-r'.

The graph is a machine independent model of the operation of a data reference macro. It is machine independent because it operates on the primitives of a data definition facility in which data for a variety of machines can be automatically expressed.

Since the INCREMENT function is machine independent it is built into DMACS. Using this function, the language designer can write his macros without worrying about how different data items map into core. In a similar manner, a CONVERT function to convert the pointer into a data item 'location' (to be input to a load /update routine) is built into DMACS. The logic for this routine is discussed in the next section.

Using these two built in routines, a language designer can write a subscript macro with the following logic:

```
SUBSCRIPT X,I
          1. subtract 1 from I yielding Value(I-1).
          2. if the element length of X is (1,0) or (0,1)
                    then INCREMENT X by Value(I-1)
             else multiply Value(I-1) by the element length of X
                    and INCREMENT X by the result
          3. if X is a terminal data item
                    then apply CONVERT to the pointer computed above.
```

(When incrementing X with a value, the units of the element length of X determines which component of the pointer is incremented. If I is a compile time value, the subtraction and multiplication can be done at compile time. Otherwise code must be generated to perform these operations at run time.)

The macro is completely free of machine dependent detail. Using the two functions built into DMACS, the language implementer is able to write a macro dealing only with the semantics of his source language. For instance, such a macro might include logic to handle subscript bounds or to handle special types of subscripting such as for triangular matrices, but need pay no concern to machine structure at all.

4.3.2 THE CONVERT FUNCTION

The CONVERT function takes a pointer in the form of a 4-tuple (CA,RA,CB,RB) as discussed in the previous section, and converts it into a form suitable for use by a load/update routine. When the pointer is expressed in addressable units and references a simply accessable item, conversion is not necessary. When the pointer includes bits, however, the bit elements of the pointer must be normalized to yield a number of addressable units, and a local bit displacement within the memory unit pointed to by

the addressable elements RA and CA.

First we discuss the problem of normalization, and then how it fits into the CONVERT logic as a whole.

NORMALIZATION: Consider the problem of accessing a bitstring on the PDP-10 and on the IBM-360 given the base address of a data area and a bit index into it. On the PDP-10, the index should be divided by 36 (bits/word), yielding a full-word index as the quotient, and the bit displacement as remainder. On the 360, assuming the load/update routine uses full word load instructions, the address of a full-word boundary is wanted, together with a bit displacement to within that word. Therefore, the index should be divided by 32 (bits/word), yielding a bit displacement as remainder. Multiplying the quotient by 4 would then yield an index in addressable units. Thus a data type may have the following attributes when implemented on a particular machine: Nd- a number to divide a bit pointer by, to yield a 'local' bit pointer as a remainder, Na- a number to multiply the result of that division by to yield addressable units.

When the bit pointer is a compile time value, this normalization is performed at compile time. Otherwise, code

must be generated to perform the normalization at run time.

CONVERT: The convert function takes a pointer P=(CA,RA,CB,RB) and converts it to a location L=(CA,RA,cbl) or L=(CA,RA,rbl), where cbl and rbl are local bit displacements into a memory unit. The logic of this function is:

1. If CB=nil and RB=nil, then normalize CB at compile time yielding ca and cbl. Then INCREMENT P by ca

2. If RB≠nil then do
   (a. if CB≠ nil then
     (generate code to add RB and CB yielding RB)
   b. generate code to normalize RB yielding ra and rbl
   c. INCREMENT P with ra)

This function yields an expression which can be input to a load/update pair. This function operates on the primitives of a data-definition facility and is therefore machine-independent.

4.4 SUMMARY

The chapter describes how a data definition facility is built into DMACS to facilitate the writing of machine independent macros. Then it discusses how this facility is used: how the machine specifier describes his machine memory, accessing functions, and the mapping of source data types into core; and how DMACS then uses the information to

compute the primitives which describe a source program's data. The chapter then discusses how machine independent macros are written in terms of two machine independent functions (INCREMENT and CONVERT), operating over these primitives. These two functions embody the substance of the machine related part of data reference macros. They are built into DMACS to be used as a tool by the language implementer.

The basic concept set forth in this chapter is the use of a data definitional facility. The rest of the chapter is built around this idea. It is instructive to ask how much more flexibility the definitional facility affords over a code generator for a single target machine. At first glance, it might appear that the definitional facility merely lets DMACS describe data with different numbers on different machines, but perform the same manipulations with those numbers in all cases. This is not true. The definitional facility gives the language implementer the ability to handle a given source data reference with different sections of his logic on different machines. Thus an array of characters can be handled for the IBM-360 as an array of addressable units with element length of 1, whereas on the PDP-10, it would be handled as an array of bitstrings

of length 9. The operations performed on these element lengths could be different, and the load update routines used to access the items could be different.

Thus the definitional facility of DMACS provides a flexible interface between machine structure and macro logic. At the same time, it is an interface that is almost invisible to both the machine specifier and the language implementer. The language implementer is able to think primarily in terms of the semantics of his language irrespective of machine structure, and the machine specifier merely gives a description of his machine. DMACS takes care of binding the macros and the machine description together.

CHAPTER V

CONCLUSIONS AND FURTHER WORK

5.1 AREAS FOR FURTHER WORK

5.1.1 FURTHER ASPECTS OF CODE GENERATION

The scope of the present research is limited since it does not address the task of making an entire compiler machine independent. Only two classes of macros are studied, and only a limited set of possible operand types are allowed. Also, many machine idiosyncrasies, such as interrupt handling, are ignored.

The problem of making a powerful compiler machine independent is a difficult and a messy one. The problem is somewhat softened by the fact that many machine idiosyncracies can properly be handled by subroutines, and thus may not prove to be insurmountable stumbling blocks.

One signifigant area not dealt with is the class of control macros, such as subroutine calls, entry and return macros, etc. These macros may not, however, require any elaborate mechanisms to allow machine independence. In general, such control macros are implemented very similarly on different machines and may be describable merely by appropriate code sequences. One minor problem is to assure

that the stack is allocated in the correct units.

Types of operands not considered include character strings and decimal operands such as those found on the IBM-360. Both of these types of operands are generally not manipulated via registers, but rather by subroutine or by special memory-memory instructions. The model of computation in Chapter 3 is oriented primarily towards manipulating values using registers. More work is also needed to determine exactly how load/update routines can best be defined to fit into a machine independent framework.

5.1.2 EXTENDING THE MODELS

The models presented in this paper are set forth primarily to isolate some basic ideas involved in code generation, and to provide a basis for more general extensions which could include a broader spectrum of machine structure.

In particular, one might relax some of the constraints imposed on register structure in Chapter 3, (perhaps to include such machines as a stack machine), and develop an automatic mechanism for attaining permitted

states in this less constrained system. By relaxing constraints in this fashion, it might be possible to obtain a number of different automatic mechanisms, together with classes of machine structures which can be handled by each mechanism.

In a similar vein, one might consider different possible addressing structures, and determine how the machine independent data reference logic can be modified to accomodate them. In particular, it might be useful to look at addressing on small machines, such as the PDP-8, which tend to have anomolous addressing strategies due to bit conserving design considerations. In fact, such machines might be practical candidates for a descriptive system like DMACS, since they tend to be reasonably similar, and since they tend to be unsuitable for sustaining compilers themselves.

5.2 SUMMARY OF RESULTS

The present research has examined the two most common types of macro used for handling arithmetic values: computation macros, and data reference macros. For each of the two types of macro, the paper develops a machine independent formalism which models the machine dependent

aspects of the macro's logic: a state machine for computation macros; and the INCREMENT and CONVERT functions for data reference macros.

Chapters 3 and 4 show how the models can be incorporated into DMACS, a descriptive macro system. A language implementer can use the models as tools, writing his macros in terms of machine independent primitives which invoke the model. A machine specifier can then describe his machine, and descriptively fill out the primitives as they apply to his machine.

Thus the research has several purposes:

1. The research is a first attempt to formalize some of the logic involved in generating code for high level languages.

2. The research is an attempt to see what is involved in attaining machine independence in a code generator, similar to the language independence and the token independence acheived by automatic parsing and automatic lexical systems.

3. Towards this end, this paper explores the question of just what might reasonably constitute a 'description' of a machine.

4. The research helps make clearer the distinction between the semantics of a high level language and the structure of a target machine, a distinction that is often unclear in a compiler oriented towards a single machine.

REFERENCES

1. Feldman, J., and Gries, D., "Translator Writing Systems", CACM, Feb. 1968

2. Crespi-Reghizzi, Stephano, "The Mechanical Acquizition of Precedence Grammars", UCLA-ENG-7054, June, 1970. School of Engineering and Applied Science, University of California, Los Angeles.

3. DeRemer, F.L., "Practical Translators for LR(k) Languages", MAC-TR-65, Oct. 1969, Project MAC, M.I.T., Cambridge, Ma.

4. Johnson, W.L., et al., "Automatic Generation of Efficient Lexical Processors Using Finite State Techniques", CACM, Dec. 1968

5. Orgass, R. J., and Waite, W.H., "A Base for a MObile Programming System", CACM, Sept. 1969

6. Waite, W.M., "Building a Mobile Programming System", Report No. 69-2, Computing Center, University of Colorado, June 1969

7. Strong, J., et al., "The Problem of Programming Communication with Changing Machines: A Proposed Solution",

CACM, Aug. 1958

8. Steel, T.B. Jr., "A First Version of UNCOL", Proceedings WJCC 1961, pp. 371-378.

9. Elson, M., and Rake, S.T., "Code-Generation Technique for Large Language Compilers", IBM Syst. J., No. 3, 1970

10. Cheatham, T., "Course Notes on Compiling", Applied Math 295, Harvard University, 1969

11. Strachey, C., "Fundamental Concepts in Programming Languages," Programming Research Group, Oxford University, England.

12. Graham, R.M., "Programming Systems" (to be published)

13. Graham, R.M., et al., "LPS- A Language Processing system", ProgLing Memo No. 1, Project MAC, M.I.T., Cambridge, Ma., 1969

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Massachusetts Institute of Technology
Project MAC

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP
None

3. REPORT TITLE
Automatic Creation of a Code Generator from a Machine Description

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Thesis, EE Degree, Department of Electrical Engineering, March 1971

5. AUTHOR(S) *(Last name, first name, initial)*
Miller, Perry L.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
May 1971	98	13

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
Nonr-4102(01)	MAC TR-85 (THESIS)
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. AVAILABILITY/LIMITATION NOTICES
Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Advanced Research Projects Agency 3D-200 Pentagon Washington, D.C. 20301

13. ABSTRACT

This paper studies some of the problems involved in attaining machine independence for a code generator, similar to the language independence and the token independence attained by automatic parsing and automatic lexical systems. In particular, the paper examines the logic involved in two areas of code generation: computation and data reference. It presents models embodying the logic of each area and demonstrates how the models can be filled out by descriptive information about a particular machine. The paper also describes how the models can be incorporated into a descriptive macro code generating system (DMACS) to be used as a tool by a language implementer in creating a machine independent code generator, which can be made machine-directed by a suitable description of a particular machine.

14. KEY WORDS
Compilers Translator Writing Systems Machine Code Generation
Programming Languages

DD FORM 1 NOV 68 1473 (M.I.T.)

UNCLASSIFIED
Security Classification